Runtime adaptation of an iterative linear system solution to distributed environments

Masha Sosonkina

Department of Computer Science, University of Minnesota, Duluth,
320 Heller Hall, 10 University Drive, Duluth, Minnesota 55812-2496
masha@d.umn.edu

Abstract. Distributed cluster environments are becoming popular platforms for high performance computing in lieu of single-vendor supercomputers. However, the reliability and sustainable performance of a cluster are difficult to ensure since the amount of available distributed resources may vary during the application execution. To increase robustness, an application needs to have self-adaptive features that are invoked at the runtime. For a class of computationally-intensive distributed scientific applications, iterative linear system solutions, we show a benefit of the adaptations that change the amount of local computations based on the runtime performance information. A few strategies for efficient exchange of such information are discussed and tested on two cluster architectures.

1 Introduction

Distributed environments are now widely used for computationally-intensive scientific applications. However, efficiency and robustness are still difficult to attain in such environments due to the varying distributed resource availability at any given time. For example, the "best effort" interconnection networks have no mechanism to satisfy application communication requirements all the time. Thus an application needs to have its own adaptive mechanisms. Applications that adjust their quality of service (i.e., the computation and communication demands) to the state of interconnecting network and computing nodes are quite common in multimedia already. In scientific computing, however, the concept of self-adaptation is rather new. For scientific applications with adaptation features, distributed environments may win a performance battle over single-vendor supercomputers, since adaptation makes scientific computing more robust and fault-tolerant.

Linear system solution is usually the most computationally-expensive part of many high-performance computing applications. Thus focusing on its adaptive features will affect significantly the overall performance of applications. Large-scale sparse linear systems are often solved using *iterative solution* techniques, which find an approximate solution given a desired accuracy. These techniques have high degree of parallelism and are easy to implement. Here, we propose a few adaptation strategies for an iterative linear system solution in distributed environments. The paper is organized as follows. Section 2 outlines a general

framework of a parallel iterative linear system solution method. Next, we describe a few strategies for obtaining run-time information about the system performance as measured by elapsed time. In section 3, numerical experiments are provided for different cluster environments. We summarize the work in Section 4.

2 Distributed iterative linear system solution with adaptation features

An iterative solution method can be easily implemented in parallel, yielding a high degree of parallelism. Consider, for example, a parallel implementation of FGMRES [4], a variation of a popular solution method, restarted Generalized Minimum RESidual algorithm (GMRES) [3]. If the classical Gram-Schmidt procedure is used in its orthogonalization phase, an iteration of the parallel algorithm has only two synchronization points, in which all-to-all processor communications are incurred. A drawback of iterative methods is that it is not easy to predict how fast a linear system can be solved to a certain accuracy and whether it can be solved at all by certain types of iterative solvers. This depends on the algebraic properties of the matrix. To accelerate the convergence of an iterative method, a linear system can be transformed into one that has the same solution but for which the iteration converges faster. This transformation process is called preconditioning. With a good preconditioner, the total number of steps required for convergence can be reduced dramatically, at the cost of a slight increase in the number of operations per step, resulting in much more efficient algorithms. In distributed environments, an additional benefit of preconditioning is that it reduces the parallel overhead, and thus decreases the total parallel execution time.

2.1 Distributed matrix representation and block-Jacobi preconditioning

One way to partition the linear system $Ax = b$ is to assign certain equations and corresponding unknowns to each processor. For a graph representation of sparse matrix, graph partitioner may be used to select particular subsets of equation-unknown pairs (subproblems) to minimize the amount of communication and to produce subproblems of almost equal size. It is common to distinguish three types of unknowns: (1) Interior unknowns that are coupled only with local equations; (2) Local interface unknowns that are coupled with both non-local (external) and local equations; and (3) External interface unknowns that belong to other subproblems and are coupled with local equations. Thus each local vector of unknowns x_i is reordered such that its subvector u_i of internal components is followed by the subvector y_i of local interface components. The right-hand side b_i is conformly split into the subvectors f_i and g_i , i.e.,

$$x_i = \begin{pmatrix} u_i \\ y_i \end{pmatrix} \; ; \quad b_i = \begin{pmatrix} f_i \\ g_i \end{pmatrix} \; .$$

When block-partitioned according to this splitting, the local matrix A_i residing in processor i has the form

$$A_i = \left(\begin{array}{c|c} B_i & F_i \\ \hline E_i & C_i \end{array} \right) , \tag{1}$$

so the local equations can be written as follows:

$$\begin{pmatrix} B_i & F_i \\ E_i & C_i \end{pmatrix} \begin{pmatrix} u_i \\ y_i \end{pmatrix} + \begin{pmatrix} 0 \\ \sum_{j \in N_i} E_{ij} y_j \end{pmatrix} = \begin{pmatrix} f_i \\ g_i \end{pmatrix} . \tag{2}$$

Here, N_i is the set of indices for subproblems that are neighbors to the subproblem i. The term $E_{ij}y_j$ reflects the contribution to the local equation from the neighboring subproblem j. The result of the multiplication with external interface components affects only the local interface unknowns, which is indicated by zero in the top part of the second term of the left-hand side of (2).

The simplest of distributed preconditioners is the block-Jacobi procedure (see, e.g., [3] and the references therein). This form of a block-Jacobi iteration, in which blocks refer to the subproblems, is sketched next.

ALGORITHM **1** *Block Jacobi Iteration:*

1. *Obtain external data* $y_{i,ext}$
2. *Compute (update) local residual* $r_i = (b - Ax)_i = b_i - A_i x_i - \sum_{j \in N_i} E_{ij} y_j$
3. *Solve* $A_i \delta_i = r_i$
4. *Update solution* $x_i = x_i + \delta_i$.

The required communication, as well as the overall structure of the routine, is identical to that of a matrix-vector multiplication. In distributed environments, the block-Jacobi preconditioner is quite attractive since it incurs no extra communication and becomes less expensive with increase in processor numbers. However, it is well-known that, in this case, the effect of block-Jacobi on convergence deteriorates requiring more iterations to converge.

2.2 Adaptive strategies

For the local preconditioning strategies, such as block-Jacobi, the amount of work each processor accomplishes in the preconditioning application is different and depends on the properties of the local submatrices. Since the properties of the local submatrices may vary greatly, the times of the preconditioning phase may also differ substantially leading to a load imbalance among processors. Load imbalance may also occur under the *unequal resource* conditions in one or more processors. We assume that the unequal resource condition arises when nodes differ in computational power either due to to their hardware configuration or due to competing loads that consume a part of node resources. For unbalanced local loads, when the processor synchronizations take place (in the orthogonalization

phase of FGMRES and during the matrix-vector product), the processors with small preconditioning workload must wait for the other processors. One way to avoid this idling is to force all the processors to spend the same time in the preconditioning application. The time may be fixed, for example, based on the time required by the processor with the largest workload to apply a preconditioning step. The rationale is that it is better to spend the time, which would be wasted otherwise, to perform more iterations in the "faster" processors. A better accuracy may be achieved in "faster" processors which would eventually propagate to others, resulting in a reduction of the number of iterations to converge.

There are several approaches to control the "fixed-time" condition for a block-Jacobi preconditioning step (Algorithm 1) when an iterative process (e.g., GMRES) is used to solve the linear system in line 3. One of these approaches (tested in [5]) is to change locally the number of inner GMRES iterations at a certain (outer) iteration of FGMRES based on some criterion. The following iteration adjustment parameters have been determined experimentally and applied after each preconditioning step in processor i, $(i = 1, \ldots, p)$:

$$\text{if } (\Delta_j^i > n_{j-1}^i/3) \quad n_j^i = n_{j-1}^i + \Delta_j^i,$$

where n_j^i is the number of the inner iterations in the (next) jth iteration of FGMRES; Δ_j^i is the number of iterations that processor i can fit into the time to be wasted in idling otherwise at the jth outer iteration of FGMRES. Specifically,

$$\Delta_j^i = \frac{(T_{\max} - T^i)n^i}{T^i},$$

where $T_{\max}$ is the maximum time among all the processors and T^i is the time for processor i to perform preconditioning operations during $j-1$ previous outer iterations; n^i is the total number of preconditioning operations performed by processor i so far. The number of inner iterations n_j^i can be updated provided that the limit n_{lim} on the number of inner iterations is not reached.

The maximum time $T_{\max}$ has been obtained by an all-to-all communication required by the global maximum computation. However, for the distributed environments in which the interconnecting network has large communication latency, obtaining $T_{\max}$ may incur significant parallel overhead. It is, therefore, desirable to use already existing communication points to exchange the timing information, so that no separate message is issued for sharing the performance information.

First, we explain how the communication during a distributed matrix-vector multiplication may be exploited. The value of T^i is appended to the message containing the local interface unknowns sent to a neighbor. Upon receipt of all $T^k (k \in N_i)$, processor i determines locally the maximum $T'_{\max}$ over all T^k and uses it in adaptation decision for the jth iteration. A disadvantage of this strategy is that the information is exchanged only among neighbors, so there is no knowledge of the global $T_{\max}$. To alleviate this drawback, the computation of Δ_j^i has been modified to use the information only from the previous $(j-1)$st iteration.

Second, a strategy that finds the global maximum time can be designed such that no extra communication takes place. An all-to-all communication of the FGMRES orthogonalization phase may be used to compute and disseminate global $T_{\max}$. Since in the distributed orthogonalization, a global sum is computed, we need to introduce a "mixed" reduction operation that determines a vector `(sum, max)`. Message Passing Interface [1], which we use as a communication library, permits such a mixed user-defined reduction operation.

3 Numerical experiments

The experiments have been performed on the two types of distributed architectures: a cluster of 8 nodes with two Power3 (200MHz) processors connected via Gigabit Ethernet and a cluster of 64 nodes, in a Single PentiumPro (200MHz) processor mode, connected via Fast Ethernet. Each node of Power3 and PentiumPro clusters has 1 GB and 256 MB of main memory, respectively. Both clusters are built in Ames Laboratory (Ames, IA). The test problem AF23560 comes from the Davis collection [2]. This problem is unstructured, has 23,560 unknowns and 484,256 nonzeros in the matrix. In the preconditioning phase, block-Jacobi is used such that the linear system solve is handled by ILUT-preconditioned GMRES(20) with the following parameters: 30 fill-in elements per row in the ILUT factorization [3], $n_0^i = 5$, $n_{\text{lim}} = 20$, and the relative accuracy of 10^{-2}. This accuracy is increased to 10^{-8} whenever n_j^i is adapted.

Figure 1 compares the time (top) and iteration (bottom), for the following implementations: standard block-Jacobi (`Jac`) and two versions of adaptive block-Jacobi (`Jac_global_1` and `Jac_local`). `Jac_global_1` uses a separate global communication to gather performance information and `Jac_local` exchanges the timing information among the neighbors only. It can be seen that both adaptive strategies are more robust than the standard algorithm since they converge when standard block-Jacobi fails (for 8 and 10 processors). With increase in processor numbers, the communication cost starts to dominate the execution time. Thus the adaptive version without extra communication point becomes beneficial. However, the neighbor-only balancing strategy is inferior to the global one so the performance of `Jac_local` is hindered.

Next, we monitor the performance of the proposed adaptive versions under the unequal resource conditions. In certain cluster nodes, competing loads are introduced such that they consume a significant (constant) amount of main memory. We choose to focus on memory resources since the performance of scientific applications is often memory-bound for large-scale problems.

The test problem and solution parameters are taken as in the previous set of experiments. The tests have been performed on four processors. A simple program consuming a constant amount (512 MB) of memory (denoted as load L1) was run on *node0* of the Power3 cluster together with the distributed linear system solution the mapping of which included *node0*. Similarly, a constant amount of memory equal to 150 MB (denoted as load L2) was consumed on *node0* of the PentiumPro cluster while a parallel linear system solution was running on

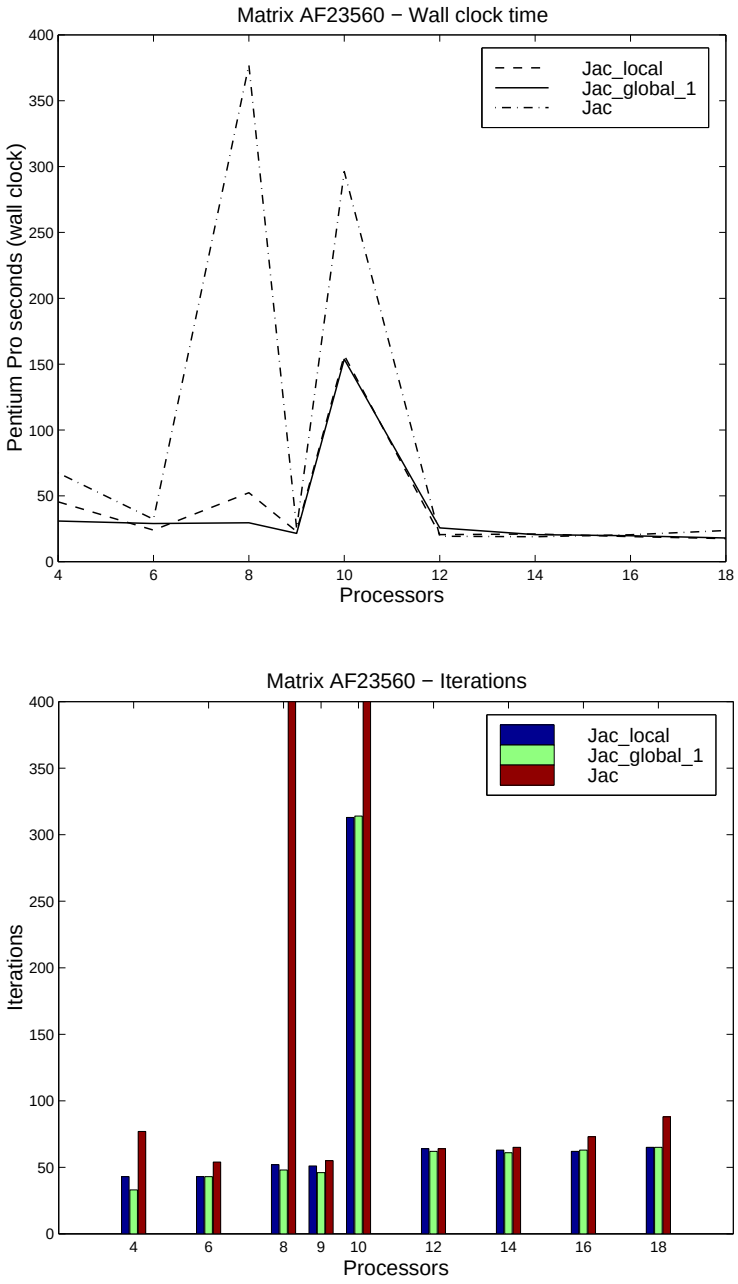

Fig. 1. Time (top) and iteration (bottom) comparisons of three variations of block-Jacobi for subproblems of varying difficulty on increasing processor numbers

a set of four nodes including *node0*. For the experiments on both clusters, Table 1 provides the number of the outer iterations `outer_it` until convergence and the total solution time `total_time` for the following variations of the solution process: standard block-Jacobi `Jac` with the number of inner iterations ${n^i}_0 = 5$, ${n^i}_0 = n_{\lim} = 20$, and with unequal subproblem sizes `ne_part` (for ${n^i}_0 = 5$); `Jac_local`; `Jac_global_1`; and (for Power3 only) `Jac_global_2`, an adaptive version of block-Jacobi that exchanges performance information via a global communication in the FGMRES orthogonalization phase. To partition a problem into unequal subproblems (the `ne_part` variation of standard block-Jacobi), a graph partitioner was modified such that it takes as an input a vector of relative sizes for each of the subproblems and attempts to produce partitions of corresponding sizes.

We observe that, in general, adapting to the unequal memory condition, either by an *a priori* matching of partition size to the extra loads or by runtime iteration adjustment, improves performance. A drawback of an *a priori* size selection, however, is that it is difficult to choose the relative partition sizes to reflect the unequal resource availability, which may change dynamically. Standard block-Jacobi with ${n^i}_0 = n_{\lim}$ appears to be quite competitive on Power3 processors, since the cost per preconditioner (inner) iteration does not increase much going from 5 to 20 iterations. All adaptive variations show comparable performance. `Jac_local` outperforms other variations on the Power3 cluster. This can be explained by the partition connectivity (Figure 2) of the problem and the dual-processor architecture of the Power3 cluster nodes. Similar situation can be seen in Table 2, which, in the column '`2 nodes / 2 L2`', gives the timing and iteration results on the PentiumPro cluster when both *node0* and *node1* have load L2. The column '`2 nodes / 3 L2`' of Table 2 shows the case when *node0* has load L2 and *node1* has 2×L2. A global strategy `Jac_global_1` appears to react better than the local strategy to varying imbalance conditions.

4 Conclusions

We have showed that parallel iterative linear system solution may need to adapt itself when the distributed resources, such as memory, are not equal among nodes in a distributed environment. Block-Jacobi preconditioning has been taken as an example to dynamically balance unequal resources with the amount of local work per processor. The dynamic adjustment of local iterations reduces the imbalances due to different resource availability and subproblem difficulty, which makes the linear system more robust and decreases the number of iterations to convergence. We observed that an *a priori* adjustment of subdomain sizes is not a viable alternative since both the environment and algorithm performances may vary dynamically. It has been shown that the runtime performance information exchange requiring no separate communications is advantageous in distributed memory environments that have high-latency interconnects.

Table 1. The performance of block-Jacobi variations when one cluster node has amount of memory depleted

Method		Power3		PentiumPro	
		outer_it	time_it	outer_it	time_it
Jac	$n^t{}_0 = 5$	77	87.47	77	51.75
	$n^t{}_0 = 20$	30	48.98	30	62.56
	ne_part	74	57.28	83	54.55
Jacobi_local		33	42.33	32	28.82
Jacobi_global_1		33	55.39	38	26.29
Jacobi_global_2		32	54.07		

Table 2. The performance of block-Jacobi variations with increasing memory imbalance in the nodes of the PentiumPro cluster

Method		2 nodes / 2 L2		2 nodes / 3 L2	
		outer_it	time_it	outer_it	time_it
Jac	$n^t{}_0 = 5$	77	66.67	77	99.56
Jacobi_local		33	35.54	32	49.21
Jacobi_global_1		35	38.11	33	48.72

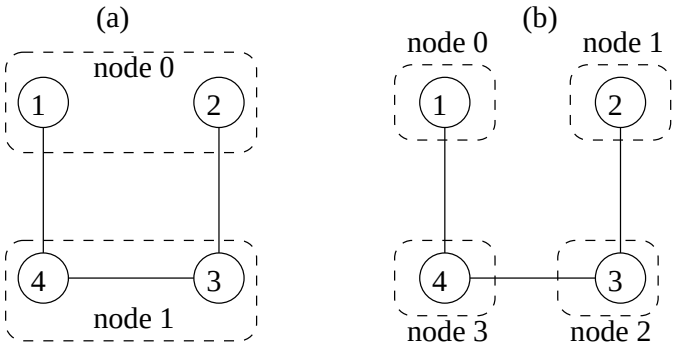

Fig. 2. Partition connectivity and mapping for the test problem solved in four processors of the Power3 (a) and PentiumPro (b) clusters

References

1. MPI Forum. MPI: A message-passing standard. *Intl. J. Supercomut. Applic.*, 8, 1994.
2. T. Davis. University of florida sparse matrix collection. *NA Digest*, 1997. `http://www.cise.ufl.edu/~davis/sparse`.
3. Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS publishing, New York, 1996.
4. Y. Saad and A. Malevsky. PSPARSLIB: A portable library of distributed memory sparse iterative solvers. In V. E. Malyshkin et al., editor, *Proceedings of Parallel Computing Technologies (PaCT-95), 3-rd international conference, St. Petersburg, Russia, Sept. 1995*, 1995.
5. Y. Saad and M. Sosonkina. Non-standard parallel solution strategies for distributed sparse linear systems. In A. Uhl P. Zinterhof, M. Vajtersic, editor, *Parallel Computation: Proc. of ACPC'99*, Lecture Notes in Computer Science, pages 13–27, Berlin, 1999. Springer-Verlag.